特典会DXサービス「アンコール」要件定義書・基本設計書 (Draft)

1. システム全体構成図 (アーキテクチャ)

フロントエンドにAstroを採用し、Vercelでホスティングすることで高速な描画を実現します。

バックエンド処理は、BaaSであるSupabaseと、重い処理（画像合成・決済処理など）を担当するGCP Cloud Runに分離します。

- **Frontend**: Astro (SSR Mode) + React or Vue (カメラ/インタラクティブ部分)
    - Deploy: Vercel (GitHub連携)
- **Backend / Auth / DB**: Supabase
    - Auth: 認証 (Fan, Idol, Admin)
    - Database: PostgreSQL (イベント情報、チケット管理、ポイント台帳)
    - Realtime: タイマー同期、通知
- **Application Logic / Worker**: Google Cloud Run
    - 画像合成処理（フレーム合成）、動画処理
    - 決済Webhook処理
    - バッチ処理（ランキング集計など）
- **File Storage**: Google Cloud Storage (GCS)
    - 撮影データ（高解像度写真、動画）、デジタルグッズ素材

---

2. 機能要件 (User Stories & Features)

提案書に基づき、3つのロール（ファン、アイドル、運営）ごとの機能を定義します。

A. ファン (User)

目的: スムーズな参加、思い出のデジタル管理、推し活の可視化

機能カテゴリ	詳細要件	実装イメージ **(Tech)**
認証・登録	新規登録、ログイン、プロフィール設定（ニックネーム等）	Supabase Auth (Email, Social)
ポイント購入	クレジットカード等でのポイントチャージ	Stripe連携 → Cloud Run (Webhook) → Supabase更新

チケット購入	ポイント消費による特典券（参加券）の入手	Supabase Transaction (Atomic処理必須)
特典会参加	保有チケット表示、QRコード表示/読み取り、待機列管理	Astro (Client Island), Supabase Realtime
撮影・保存	Webカメラ起動、タイマー表示、撮影、自動アップロード	WebRTC/MediaDevices API → GCS (Signed URL)
アルバム	撮影した写真・動画の閲覧、ダウンロード、SNSシェア	Astro (Image Optimization), GCS
通知受取	アイドルからの事後メッセージ（画像/ボイス）受信通知	Supabase Realtime / Web Push
ランキング	自身のランク確認、推しへの貢献度確認	Supabase View / Cloud Run (集計)

B. アイドル/メンバー (Cast)

目的: ファン管理、安全な特典会実施、ファンサービス向上

機能カテゴリ	詳細要件	実装イメージ **(Tech)**
認証	運営が発行したアカウントでのログイン	Supabase Auth
もぎり・撮影	ファンのチケット確認（QR読取/タップ）、撮影モード起動	QR Scanner Lib, WebRTC
顧客情報表示	撮影相手のニックネーム、ランク、過去来場回数の表示	Supabase Query (RSLで制御)

特典付与	撮影後の写真への「落書き」「ボイス録音」送信	Canvas API (描画), Web Audio API (録音) → GCS
ファン管理	「未対応リスト（メッセージ未送信）」の管理、ファン一覧	Astro SSR + Supabase
売上確認	個人の売上、ランキング確認	Supabase (Aggregation)

C. 運営事務所 (Admin)

目的: 売上最大化、不正防止、業務効率化

機能カテゴリ	詳細要件	実装イメージ (Tech)
イベント管理	イベント作成、特典券種別設定（価格、時間、枚数）	Supabase (CRUD)
売上管理	リアルタイム売上集計、出金申請	Dashboard (Recharts等で可視化)
顧客分析	ファンの属性分析、頻度分析、LTV計測	Cloud Run (BigQuery連携も視野)
デジタルグッズ	フォトフレーム、デジタルコンテンツの登録・販売	GCSへのアセットアップロード
一斉連絡	プッシュ通知配信、会場での呼び出し機能	Cloud Run (FCM等) or Supabase Realtime

---

3. データベース設計 (Supabase Schema Draft)

Supabase (PostgreSQL) の主要なテーブル構成案です。

SQL
```
-- ユーザー（全ロール共通管理、metadataでロール識別）
create table profiles (
```

```
  id uuid references auth.users not null,
  role text check (role in ('fan', 'idol', 'admin')),
  nickname text,
  avatar_url text,
  points_balance integer default 0, -- 重要: ポイント残高
  rank_score integer default 0,
  created_at timestamp with time zone
);

-- イベント
create table events (
  id uuid primary key default uuid_generate_v4(),
  organizer_id uuid references profiles(id),
  title text not null,
  event_date timestamp with time zone,
  location text,
  status text -- scheduled, active, completed
);

-- 特典券商品マスタ
create table ticket_products (
  id uuid primary key default uuid_generate_v4(),
  event_id uuid references events(id),
  idol_id uuid references profiles(id),
  title text, -- "2ショットチェキ" 等
  price_points integer not null,
  duration_seconds integer not null, -- 撮影時間
  stock_limit integer -- 販売上限
);

-- 購入済みチケット（インベントリ）
create table user_tickets (
  id uuid primary key default uuid_generate_v4(),
  user_id uuid references profiles(id),
  ticket_product_id uuid references ticket_products(id),
  status text default 'valid', -- valid, used, expired
  used_at timestamp with time zone
);

-- 撮影データ（成果物）
create table media_assets (
  id uuid primary key default uuid_generate_v4(),
  user_id uuid references profiles(id), -- ファン
  idol_id uuid references profiles(id), -- アイドル
  event_id uuid references events(id),
  original_url text, -- GCS path (撮影生データ)
  decorated_url text, -- GCS path (落書き済み)
  voice_message_url text, -- GCS path
```

```
  media_type text, -- photo, video
  status text default 'pending_review' -- pending, published
);
```

---

4. 技術的な重要ポイントと実装戦略

ご提示のスタックにおける具体的な実装アプローチです。

① フロントエンド (Astro + Vercel)

- **Hybrid Rendering:** 基本はSSG/SSRで高速表示し、カメラ機能やチャット機能など動的な部分は client:only="react" (またはVue/Svelte) でアイランドアーキテクチャを採用します。
- **PWA化:** ネイティブアプリのような体験（ホーム画面追加、フルスクリーンカメラ）を提供するため、PWA (Progressive Web App) として実装します。vite-pwa プラグインを使用。
- **カメラ機能:** ブラウザの getUserMedia APIを使用しますが、iOS Safari等での挙動差異を吸収するため、入念なテストが必要です。

② バックエンド処理 (Cloud Run vs Supabase Functions)

- **Cloud Runの役割 (Python/FastAPI または Node.js):**
  - **画像合成:** アイドルが描いたCanvasデータを、サーバーサイドで高画質に元の写真と合成・焼き付けを行う処理は、Supabase Edge Functions（Deno）よりもCloud Run（Python + Pillow/OpenCV 等）の方がライブラリ資産が豊富で実装しやすいです。
  - **動画変換:** 撮影した動画のエンコード処理（ffmpeg等）が必要な場合もCloud Runが適しています。
- **Supabaseの役割:**
  - **Database & Auth:** データの主要な保存先。
  - **Realtime:** 「もぎり」をした瞬間にタイマーをファン・アイドルの両端末で同期させる処理に使用します。
  - **Storage (Wrapper):** GCSへの直接アクセスではなく、Supabase Storage (S3互換) を通すか、あるいはGCSのSigned URL発行APIをCloud Runに用意します（セキュリティ要件による）。今回は「重たい画像はGCS」との指定なので、Cloud Run経由でSigned URLを発行し、フロントエンドからGCSへ直接アップロードする方式がサーバー負荷的に最適です。

③ 非機能要件 (Security & Performance)

- **決済セキュリティ:** ポイント購入はStripe等の外部決済プロバイダを利用し、カード情報は非保持化します。
- **RLS (Row Level Security):** Supabase側で厳格なアクセス制御を行います。「自分の購入した写真しか見れない」「自分が参加したイベントしか書き込めない」等をDBレベルで強制します。
- **画像最適化:** GCSにアップロードされた高解像度画像は、Cloud Run等でサムネイル生成し、一覧表示のパフォーマンスを確保します。

---

5. 次のアクション（開発フェーズ案）

提案書の「Vol 1.5」という段階を踏まえ、まずはMVP（実用最小限の製品）を目指すスケジュール案です。

1. **Phase 1:** コア機能プロトタイプ **(2**週間**)**
    - Astro + Supabaseの環境構築
    - ユーザー登録・ログイン
    - カメラ撮影機能の**PoC**: ブラウザでの撮影、GCSへのアップロード、画像表示までの一連の流れ。
2. **Phase 2:** チケット・ポイント機能 **(3**週間**)**
    - DB設計確定
    - イベント作成画面
    - ポイント購入（テスト決済）、チケット交換
3. **Phase 3:** 特典会フロー実装 **(3**週間**)**
    - QR読み取り → タイマー起動 → 撮影 → 保存のUX実装
    - リアルタイム同期の実装
4. **Phase 4:** アイドル側機能・落書き機能 **(2**週間**)**
    - Canvasを使ったお絵かき機能
    - メッセージ送信機能

ユーザー様への確認事項

開発を進めるにあたり、以下の点について方針があれば教えてください。

1. 言語選定**:** Cloud Run上のバックエンド言語は、Python（画像処理に強い）でよろしいでしょうか？（前述のプロファイルよりPythonがお得意とお見受けしました）
2. 決済**:** ポイント購入の決済代行会社（Stripe, Pay.jpなど）の想定はありますか？
3. ライブ配信**:** 提案書に「リモート撮影」等の記載がありましたが、フェーズ1では「現地での対面デジタル撮影」にスコープを絞ってよろしいでしょうか？

この構成でよろしければ、詳細な**DB**定義書（**ER**図）や**API**定義の作成に進みます。いかがでしょうか？

1. 推奨するディレクトリ構成 **(Monorepo)**

リポジトリのルート直下で明確にフォルダを分けます。

Plaintext

```
my-app-repo/
├── frontend/          # Astro (Vercel連携)
│   ├── src/
│   ├── public/
│   ├── astro.config.mjs
│   └── package.json
```

```
│
├── backend/          # Python (GCP Cloud Run)
│   ├── app/
│   ├── Dockerfile    # Cloud Run用のビルド設定
│   ├── requirements.txt
│   └── main.py
│
├── docs/             # 設計書や要件定義書（今回アップロードしたPDFなど）
│
└── .gitignore        # 全体共通の除外設定
```

---

2. 一緒にするメリット（モノレポ） vs 分けるメリット（ポリレポ）

比較項目	一緒にする **(Monorepo)**	分ける **(Polyrepo)**
開発体験	◎ 高い フロントとバックを同時に修正する際、1つのVS Codeウィンドウで完結します。	△ 低い 2つのウィンドウを開き、それぞれのターミナルを行き来する必要があります。
コミット管理	◎ 整合性が保てる	× ズレる

	「APIの変更」と「それを使う画面の変更」を1つのコミット（PR）で管理でき、ロールバックも容易です。	バックエンドのPRをマージした後、フロントエンドのPRをマージする…という調整が必要です。
CI/CD連携	○ 設定で制御可能 VercelやGCPの設定で「特定のフォルダが変更された時だけデプロイ」が可能です。	◎ シンプル リポジトリにPush＝デプロイ、となるので設定は単純です。
ドキュメント	◎ 一元管理 API仕様書や設計書を同じ場所に置けるので、情報の散逸を防げます。	△ 分散する どちらのリポジトリに仕様書を置くか迷います。

今回のプロジェクトは「特典会アプリ」という一つのプロダクトなので、フロントとバックは密接に関係しています。「**API**を変えたら画面も変える」頻度が高いため、一緒の方が圧倒的に楽です。

---

3. Vercel と GCP の連携設定ポイント

1つのリポジトリにしても、「フロントを触ったのにバックエンドがデプロイされる（またはその逆）」という無駄を防ぐ設定が重要です。

A. Vercel 側の設定 (Frontend用)

Vercelの管理画面で、プロジェクト設定（Project Settings）を変更します。

- **Root Directory**: frontend を指定します。
  - これでVercelは backend フォルダの中身を無視して、Astroアプリとしてビルドしてくれます。
- **Ignored Build Step**:
  - Vercelには「特定のファイルが変わった時だけビルドする」設定があります。
  - Gitの設定で git diff --quiet HEAD^ HEAD ./ のようなコマンドを指定し、backend/ の変更時はビルドをスキップさせることができます。

B. GCP Cloud Build 側の設定 (Backend用)

cloudbuild.yaml（またはトリガー設定）で、バックエンドの変更のみを監視します。

- トリガーの構成:
    - 「含まれるファイル（Included files）」フィルターに backend/** を設定します。
    - これで、frontend/ 内の変更（デザイン修正など）でCloud Runが無駄にデプロイされるのを防げます。
- ビルドコンテキスト:
    - Docker buildを実行する際、カレントディレクトリを backend に指定します。